

A Framework for the Evaluation of Worst-Case System Efficiency

M. Callisto De Donato, M.R. Di Berardini

Scuola di Scienze e Tecnologie, Sezione Informatica. Università di Camerino

{massimo.callisto, mariarita.diberardini}@unicam.it

Abstract

In this paper we present FASE (**F**ast **A**synchronous **S**ystems **E**valuation), a tool for evaluating worst-case efficiency of asynchronous systems. This tool implements some well-established results in the setting of a timed CCS-like process algebra: PAFAS (a Process Algebra for Faster Asynchronous Systems). Moreover, we discuss some new solutions that are useful to improve the applicability of FASE to concrete meaningful examples. We finally use FASE to evaluate the efficiency of three different implementations of a bounded buffer and compare our results with previous ones obtained when the same implementations have been contrasted according to an efficiency preorder.

1 Introduction

In concurrent and distributed systems, study time aspects at an early stage of software development plays an important role to ensure the correct temporal execution of system activities. In recent years, PAFAS has been proposed as a powerful tool for evaluating the worst-case efficiency of asynchronous systems [6, 5]. PAFAS is a CCS[9]-like timed process algebra where system activities are represented by durationless actions and time passes in between them [2]. Thus, actions are atomic and instantaneous but have associated a time bound interpreted as the maximal time delay for their execution. This timing information can be used to evaluate efficiency without influence functionality (which actions are performed). So, compared to CCS, also PAFAS treats the full functionality of asynchronous systems. In [6], process are compared via a variant of the testing approach developed in [7] by considering test environments (as in [7]) together with a time bound. A process is embedded into the environment (via parallel composition) and satisfies a test if success is reached before the time bound in *every* run of the composed system, i.e. *even in the worst case*. This gives rise to a faster-than preorder relation over processes that is naturally an *efficiency preorder*. Furthermore, this preorder can be characterised as inclusion of a special kind of refusal traces which provide decidability of the testing preorder for finite state processes. The faster-than preorder has been equivalently defined in [5] on the basis of a performance function that gives the worst-case time needed to satisfy any test environment (or user behaviour). Another key result in [5] shows that, whenever the above testing scenario is adapted by considering only test environments that want n task to be performed as fast as possible, the performance function is *asymptotically linear*. This function is a *quantitative* performance measure that describes how fast a system responds to requests from the environment. This paper presents FASE, a corresponding tool that supports us to automatically evaluate the worst-case performance of a PAFAS process. FASE has been successfully used in [3] to relate three different implementations of bounded buffer: Fifo (first-in-first-out queue), Pipe (sequence of cells

connected end-to-end) and Buff (an array used in a circular fashion). The results obtained in [3] were also compared with those in [4] where the same implementations have been contrasted via the efficiency preorder in [6].

2 PAFAS

We adopt the following notation: \mathbb{A} (ranged over by a, b, c, \dots) is an infinite set of basic actions with the special action ω reserved for observes (test processes) in the testing scenario to signal the success of a test. Action τ represents an internal activity unobservable for other components; we define $\mathbb{A}_\tau = \mathbb{A} \cup \{\tau\}$ where elements are ranged over by α, β, \dots . We assume that actions in \mathbb{A}_τ can let time 1 pass as maximal delay before their execution; after that time they become *urgent*. The set of urgent actions is denoted by $\underline{\mathbb{A}}_\tau = \{\underline{a} \mid a \in \mathbb{A}\} \cup \{\underline{\tau}\}$ and is ranged over by $\underline{\alpha}, \underline{\beta}, \dots$. \mathcal{X} (ranged over by x, y, z, \dots) is the set of process variables, used for recursive definitions. A *general relabelling function* $\Phi : \mathbb{A}_\tau \rightarrow \mathbb{A}_\tau$ is such that $\{\alpha \in \mathbb{A}_\tau \mid \emptyset \neq \Phi^{-1}(\alpha) \neq \{\alpha\}\}$ is finite and $\Phi(\tau) = \tau$. General relabelling functions subsume both relabelling and hiding (see [6]).

The set \mathbb{P} of (*timed*) *processes* is the set of closed (i.e., without free variables) and guarded (i.e., variable x in a recursive term $\mu x.P$ only appears within the scope of a action-prefix) terms generated by the following grammar: $P ::= 0 \mid \gamma.P \mid P + P \mid P \parallel_A P \mid P[\Phi] \mid x \mid \mu x.P$, where γ is either α or $\underline{\alpha}$ for some $\alpha \in \mathbb{A}_\tau$, Φ is a general relabelling function, $x \in \mathcal{X}$ as expected and $A \subseteq \mathbb{A}$ possibly infinite. 0 is the Nil-process which cannot perform any action but may let time pass without limit. $\alpha.P$ and $\underline{\alpha}.P$ is the (action-) prefixing, known from CCS; $a.P$ can either perform a immediately or can idle for time 1 and become $\underline{a}.P$. In the latter case, the idle-time has elapsed and a must either occur or be deactivated (in a choice-context) before time may pass further. Our processes are *patient*: as a stand-alone process, $\underline{a}.P$ has no reason to wait; but as a component in $\underline{a}.P \parallel_{\{a\}} a.Q$, it has to wait for synchronisation on a and this can take up to time 1, since component $a.Q$ may idle this long. $P_1 + P_2$ models the choice between two processes P_1 and P_2 . $P_1 \parallel_A P_2$ is the parallel composition of two processes P_1 and P_2 that run in parallel and have to synchronise on A [8].

The temporal behaviour is given by means of the so-called *refusal traces*. Intuitively, a refusal trace records, along a computation, which actions P can perform ($P \xrightarrow{\alpha}_r P'$, $\alpha \in \mathbb{A}_\tau$) and which actions P can refuse to perform ($P \xrightarrow{X}_r P'$, $X \subseteq \mathbb{A}$).¹ A transition $P \xrightarrow{X}_r P'$ is a *conditional time step*. Actions in X are not urgent and, hence, P is justified in not performing them and performing a time step instead. Since other actions might be urgent, P might actually be unable to refuse any possible action (e.g. $\underline{a}.P$ can never refuse a). Nevertheless, as a components of a larger system, it can refuse some of its urgent actions due to synchronisation with the environment. As an example: as a component of $\underline{a}.P \parallel_{\{a\}} a.Q$, $\underline{a}.P$ can refuse a since its synchronisation partner Q can do so. We say that P perform a *full time step* (written $P \xrightarrow{1}_r P'$) if $P \xrightarrow{\mathbb{A}}_r P'$. A *discrete trace* is any sequence in $v \in (\mathbb{A}_\tau \cup \{1\})^*$ that P can perform. Finally, $\text{DL}(P)$ and $\text{RT}(P)$ are the sets of discrete traces and refusal traces (resp.) of P .

The efficiency preorder in [6] is timed variation of the testing preorder in [7]. In [6], (timed) tests are pairs (O, D) where O is a test environment (or user behaviour, i.e. a process that contains ω) and $D \in \mathbb{N}_0$ is an upper time bound. A process P satisfies a timed test (O, D) if each discrete trace $v \in \text{DL}(P \parallel_{\mathbb{A} \setminus \omega} O)$ whose duration (i.e. its number of 1's) is greater than D contains some ω . We say that P is faster than Q (written $P \sqsubseteq Q$) if P *d-satisfies all tests* that Q *d-satisfies*. Moreover, \sqsubseteq can be characterised by inclusion of refusal traces. This efficiency preorder is qualitative in the sense that a test is either satisfied or not, and that a process is more efficient than another or not. However, as shown in [5], it can be

¹We omit here the (almost standard) SOS-rules defining the transition relations $\xrightarrow{\alpha}$ and \xrightarrow{X}_r (see [6] for further details).

rephrased in terms of a (quantitative) *performance function* $p(P, O)$ that gives the worst-case time that P needs to satisfy the test O . In more details, $P \sqsupseteq Q$ iff $p(P, O) \leq p(Q, O)$ for all test process O . Yet, the performance function (as the preorder \sqsupseteq) contrasts processes w.r.t. any possible test environments. In some cases this might be too demanding and one can make some reasonable assumption about the user behaviours. In particular, one could be interested in users that have a number of requests (made via an *in*-action) that they want to be answered (via an *out*-action) as fast as possible. This is the class of users $\mathcal{U} = \{U_n \mid n \geq 1\}$ where $U_1 \equiv \underline{in.out}.\omega$ and $U_n = U_{n-1} \parallel_{\{\omega\}} \underline{in.out}.\omega$ (for any $n > 1$). Given these users, the *response performance* is defined to be the function $rp : \mathbb{N} \rightarrow \mathbb{N}_0$ such that $rp_P(n) = p(P, U_n)$ (n is the number of requests of the user).

Below we briefly describe how this response performance function is calculated in [5]. To this aim we only consider the so-called *response processes*, i.e. processes that can reasonably serves users in \mathcal{U}^2 . Now, we first observe that, for any given n , $rp_P(n)$ is obtained as the supremum of durations of all discrete traces in $DL(P \parallel U_u)$ that do not contain ω . Traces in $DL(P \parallel U_u)$ are just paths in $RTS(P \parallel U_u)$ that only contain full time steps. Moreover, for each of such paths there is a corresponding path in $rRTS(P)^3$ with the same number of conditional time steps. Thus, to calculate $rp_P(n)$ it will suffice to consider path in $rRTS(P)$. A first result in [5] states that the response performance of a response process P is the supremum of the number of time steps taken over all paths in $rRTS(P)$ with enough *in*'s and *out*'s to satisfy the user U_n (so called *n-critical paths*). At this stage, a key observation is that, when the number n of requests is large compared to the number of processes in $rRTS(P)$, a *n-critical* path with many time steps must contain cycles. Thus, it turns out to be essential to find the worst cycles. In [5] these worst-cycles are distinguished to be either *catastrophic* or *bad* cycles. A cycle in $rRTS(P)$ is said to be catastrophic if it has a positive number of time steps but no *in*'s and no *out*'s. More intuitively, if $rRTS(P)$ contains a catastrophic cycle, there is at least a path in $rRTS(P)$ with arbitrarily many time steps and, hence, there is at least an n such that $rp_P(n) = \infty$. If P is free from such cycles, $rp_P(n) = an + \Theta(1)$ is asymptotically linear (see Theorem 16 in [5]). The *asymptotic factor* a of $rp_P(n)$ is determined by considering cycles reached from P by a path where all time steps are full and which themselves contain only time steps that are full; let the *average performance* of such a cycle be the number of its full time steps divided by the number of its *in*'s. We call a cycle *bad* if it is a cycle of maximal average performance in $rRTS(P)$. Finally, the asymptotic factor of P is the average performance of a bad cycle.

3 Performance evaluation with FASE

FASE is a useful tool developed at University of Camerino to automatically evaluate the worst-case efficiency of asynchronous systems. It is written in Java and consists of two main components. The former one is the parser unit that reads a string representing a PAFAS process P and builds its $RTS(P)$. The second component is the performance unit that uses the $RTS(P)$ to implements all the technical stuffs discussed in the previous section. Moreover, it also provides some diagnostic informations that help the user to better understand to behaviour of the process.

The tool automatically checks if a process has some catastrophic cycles or not. The original solution proposed in [5] makes use an algorithm whose a complexity is $\theta(N^3)$ (N are the nodes of the graph $rRTS(P)$ of a process P). If P is a complex process, the state space of $rRTS(P)$ can be very large and the original solution becomes slow. FASE (see [3]) adopts a new solution that takes advantage from the correspondence between cycles and strongly connected components [1]. This improved solution has a

²In [5] a response process is a process that only perform *in*'s and *out*'s as visible actions and never produce more responses than requests.

³This is a reduced version of $RTS(P)$. See [5] for more details.

complexity of $O(N + E)$ where N and E are the nodes and the edges in $\text{rRTS}(P)$. We refer to [3] for a running time comparison between the two algorithms.

If P does not have catastrophic cycles, FASE looks for bad cycles in order to determine its average performance. In doing that, FASE adopts the original solution [5] with some improvements that provide the user with information about the bad cycle just computed. Since bad cycles are computed in $O(N^3)$, we are currently investigating new strategies to limit in some way this complexity.

We are also working on a solution to determine the response performance of P for a given n . Different approaches are under investigation but they still need to be validated. Currently, FASE executes an exhaustive search on $\text{rRTS}(P)$ that looks for the n -critical path whose duration is maximal; clearly as n increases this solution becomes soon intractable, especially for complex processes.

4 A Case Study and concluding remarks

In [3], FASE has been used to evaluate the worst-case efficiency of three different implementations of a bounded buffer of capacity $N + 2$ with $N \in \mathbb{N}^+$. These implementations have already been considered in [4]. We were interested in studying if the results stated in [4] still hold in our qualitative setting. Fifo is a bounded-length first-in-first-out queue, purely sequential and without overhead (in terms of internal actions). Pipe implements the buffer as the concatenation of $N + 2$ cells, where each one is an I/O device that stores at most one value. Cells are connected end-to-end that is the output of a cell is the input of the next one. Finally, Buff uses N cells as a storage Mem that interacts with a centralised buffer controller BC; BC manages Mem in a circular fashion and also retains the oldest undelivered value and outputs it whenever possible. In [3] we have obtained interesting results relating the three buffers. We have used FASE to prove that none of these implementations has catastrophic cycles. Moreover, we have also shown that $rp_{\text{Fifo}}(n) = 2n$, $rp_{\text{Pipe}} = 2n + N + 1$ and $rp_{\text{Buff}}(n) = 4n$. Thus, Fifo is more efficient than both Pipe and Buff, while Buff is more efficient than Pipe iff $n \leq \lfloor N + 1/2 \rfloor$. These results are quite different from those in [4] where the buffers have been compared by means of the efficiency preorder in [6]. The authors proved that Fifo and Pipe (but also Buff and Pipe) are unrelated (i.e. the former process is not more efficient than the latter and vice versa) while Fifo is more efficient than Buff but not vice versa. Intuitively, this is due to the fact that rp contrasts processes w.r.t. to a specific class of user behaviours while the preorder \sqsubseteq contrasts process w.r.t. any possible test. To prove if our intuition is correct, we are working on the definition (and characterisation) of a slight variation of the faster-than preorder given in [6] that allows us to contrast processes only w.r.t. user behaviours by some variant of refusal trace inclusion. Moreover, it still remains to investigate in which extent the approach described in [5] to other possible scenarios and to a different (maybe larger) class of tests. For what concerns FASE, a first important result achieved in [3] is the improvement of the catastrophic cycles detection since ensuring their absence is the basis for any further performance analysis. For bad cycles, we are obtaining encouraging results but they are still under validation. Moreover, it's still open the problem of finding the n -critical path for a given n ; we believe that further studies on the characteristics of an n -critical path can help us to find a useful solution.

References

- [1] A. V. Aho, J. E. Hopcroft, J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.
- [2] A. Aldini, M. Bernardo and F. Corradini. *A Process Algebraic Approach to Software Architecture Design*. Springer Publishing Company, 2009.

- [3] F. Buti, M. Callisto De Donato, F. Corradini, M. R. Di Berardini and W. Vogler. *Evaluating the Efficiency of Asynchronous Systems with FASE*. In pre-proc. of the 1st Int. Workshop on Quantitative Formal Methods, pp.101-106, Technische Universiteit Eindhoven, 2009
- [4] F. Corradini, M. R. Di Berardini and W. Vogler. *PAFAS at Work: Comparing the Worst-Case Efficiency of Three Buffer Implementations*. Asia-Pacific Conference on Quality Software 0:0231 (2001)
- [5] F. Corradini and W. Vogler. *Measuring the performance of asynchronous systems with PAFAS*. Theor. Comput. Sci. 335:187–213 (2005)
- [6] F. Corradini, W. Vogler and L. Jenner. *Comparing the worst-case efficiency of asynchronous systems with PAFAS*. Acta Inf. 38(11/12): 735-792 (2002)
- [7] R. De Nicola and M.C.B. Hennessy. *Testing equivalence for processes*. Theoretical Comput. Sci., 34:83-133, 1984.
- [8] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [9] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.